Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 180 (2007) 123–144
www.elsevier.com/locate/entcs

ReSpecT Nets: Towards an Analysis Methodology for ReSpecT Specifications

Mirko Viroli and Andrea Omicini

DEIS, Alma Mater Studiorum, Università di Bologna
via Venezia 52, 47023 Cesena, Italy
e-mail: {mirko.viroli,andrea.omicini}@unibo.it

Abstract

A key feature for infrastructures providing coordination services is the ability to define the behaviour of coordination abstractions according to the requirements identified at design-time. We take as a representative for this scenario the logic-based language ReSpecT (Reaction Specification Tuples), used to program the reactive behaviour of tuple centres. ReSpecT specifications are at the core of the engineering methodology underlying the TuCSoN infrastructure, and are therefore the "conceptual place" where formal methods can be fruitfully applied to guarantee relevant system properties.
In this paper we introduce ReSpecT *nets*, a formalism that can be used to describe reactive behaviours that can succeed and fail, and that allows for an encoding to Petri nets with inhibitor arcs. ReSpecT nets are introduced to give a core model to a fragment of the ReSpecT language, and to pave the way for devising an analysis methodology including formal verification of safety and liveness properties. In particular, we provide a semantics to ReSpecT specifications through a mapping to ReSpecT nets. The potential of this approach for the analysis of ReSpecT specifications is discussed, presenting initial results for the analysis of safety properties.

Keywords: Tuple spaces, ReSpecT tuple centres, Petri nets, Formal analysis

1 Introduction

There is an apparent dichotomy in the engineering of today software systems. On the one hand, the growing social impact of software systems on many critical aspects of human life makes the request of predictability of systems almost inescapable. On the other hand, many factors such as their complexity (encompassing not only their articulation as runtime systems, but also the intricacies of their design and development) make them mostly unpredictable in nature [25]. However, this does not prevent systems to be designed so as to exhibit some predictable behaviour—such as achieving some goals, or preventing some dangerous paths. In particular, when the components of a system are adequately uncoupled from design down to deployment stage, and suitably expressive abstractions are exploited to embody and encapsulate critical system behaviours, then the analysis and verification of partial

doi:10.1016/j.entcs.2006.10.049

properties of systems come to be not only feasible, but even central to system engineering.

This is typically the case of coordination abstractions such as blackboards, channels, or tuple spaces, that in today software systems are likely to be provided by distributed software infrastructures as services to system components [24]. By governing interaction among components, coordination abstractions (as both design and runtime abstractions) are typically in charge of critical system behaviours, such as composing individual components' activities to achieve global system goals, or preventing erroneous or malicious operations from components.

On the one hand, coordination abstractions are required to be expressive enough to capture the widest range of coordination problems possible. On the other hand, formal tools typically fall short when expressiveness grows, and tend to limit expressiveness as a pre-condition to provide meaningful results.

A subtle and refined work is then required. The main issue in the engineering of software systems is no longer to simply find out the most expressive coordination abstractions to be exploited in the design and development of complex software systems. Instead, the problem is now to devise the best compromise between the ability of an abstraction to capture the most complex coordination problems, and the availability of formal frameworks and tools enabling some forms of prediction over its behaviour—which typically *a priori* limits the abstraction expressiveness.

In this paper we start from the ReSpecT logic-based language for the specification of the behaviour of coordination abstractions [16], which was conceived and designed mainly according to expressiveness criteria [8]. In particular, ReSpecT is used to program *tuple centres* [17] in the TuCSoN coordination infrastructure [18]. Tuple centres are basically LINDA tuple spaces [13] enhanced with reactive programmable behaviours, and which extend their ability to capture the widest range of coordination problems possible. Then, we try to couple ReSpecT with Petri nets [19], which represent one of the most effective approaches to the analysis and verification of properties in the field of distributed systems. There, most of the well-known results mandate for limits to expressiveness—so that, for instance, extending Petri nets with inhibitor arcs increases the class of systems that can be modelled, but also reduces the number and sort of properties that can be tested and verified.

Paper Outline

Reconciliation between expressiveness of ReSpecT and foundation provided by Petri nets is achieved as follows.

In Section 2, we define ReSpecT *nets*, a graphic and operational formalism resembling Petri nets, which can be used to represent reactive behaviours affecting a dataspace and featuring a success/failure semantics. We show that ReSpecT nets can be encoded in terms of Petri nets with inhibitor arcs [3], thus enabling exploitation of the well-known properties and tools for the analysis of Petri nets [14,5].

Then, in Section 3 we sketch the ReSpecT language to program tuple centres, describing its syntax and operational semantics, and discussing a basic application

example to allow the reader to better grasp its peculiar programming style.

Section 4 bridges the gap between ReSpecT specifications and ReSpecT nets, paving the way toward an analysis methodology for ReSpecT specifications. In particular, we identify a fragment of the ReSpecT language, denoted ReSpecT_g^{1t}, where *(i)* only ground specifications are allowed—that is, neglecting logic variables and unification—and *(ii)* reactions feature the *one-testing* property—they are structured so as not to test for the presence of exactly n occurrences of a tuple, with $n > 1$. Then, specifications in this fragment are shown to allow for an automatic encoding into a ReSpecT net modelling precisely the same set of possible computations.

Section 5 provides some results and sketches some research directions toward the analysis of ReSpecT nets, focusing on the safety properties that can be expressed in terms of *control state reachability*, also known as *covering* [12,23]. Most notably, we show that ReSpecT specifications that do not exploit the `no_r` (test for absence) primitive are encoded into Petri nets where transitions featuring inhibitor arcs can be simply removed without altering covering properties. Petri nets without inhibitor arcs are intrinsically easier to analyse—other than covering also termination, boundedness and inevitability are decidable and can be actually verified [12].

Section 6 presents an example application of our methodology, and Section 7 concludes by providing for final remarks.

2 ReSpecT nets

We introduce a core, abstract model of ReSpecT specifications called ReSpecT nets: a formalism—equivalent in expressiveness to Petri nets with inhibitor arcs—which is shown to be able to model the computations of tuple centres programmed with a fragment of ReSpecT specifications.

2.1 Formal Model

In the following, given a set X we let it be ranged over by meta-variable x and its decorations x', x_0, and so on. Symbol $\overline{X}$ is used for the set of multisets over X, ranged over by $\overline{x}$: its elements are of the kind $x||x'||..||x^{(n)}$ (possibly with repetitions), and $\oslash$ is used for the void multiset.

Resembling Petri nets, a ReSpecT net is a structure $\langle D, F, In, Out, Inh\rangle$. D is the set of *data-places* and F is the set of *firing-places* (disjoint from D). Differently from Petri nets, transitions are not defined by a set of their own, but rather, there is exactly one transition for each firing-place—hence we sometime refer to a firing place as a transition and vice versa. Then, $In \subseteq F \mapsto \overline{D}$ represents the *incoming arcs*, associating to each transition $f \in F$ the multiset of data-places $\overline{d} \in \overline{D}$ which are sources of the arcs; $Out \subseteq F \mapsto (\overline{D} \cup \overline{F})$ represents the *outgoing arcs*, associating to each transition the multiset of data- and firing-places which are targets of the arcs; and $Inh \subseteq F \mapsto \overline{D}$ represents the *inhibitor arcs*, from data-places to transitions. Given an element $f \in F$, we assume that *(i)* no data-place is the source for both an incoming and an inhibitor arc—that is, $In(f) \cap Inh(f) = \oslash$—, and that *(ii)* no place is the source for more than one inhibitor arc towards the same transition—that is,

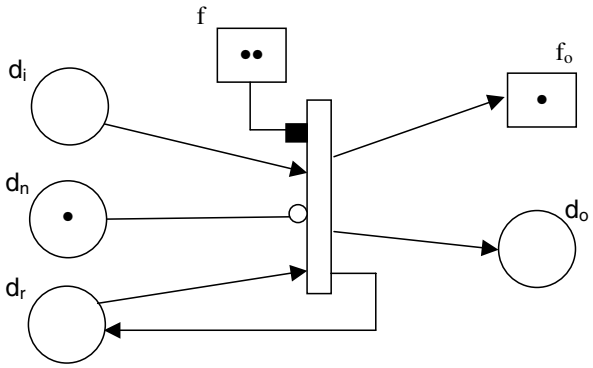

Fig. 1. Notation for ReSpecT nets

$d||d \not\subseteq Inh(f)$ for each d.

Similarly to Petri nets, the state of a ReSpecT net at a given time is a *marking* over data- and firing-places that describes how many *tokens* reside in each such place, namely, a function mapping data- and firing-places to natural numbers. Equivalently, this is described as an element $\langle \overline{d}, \overline{f} \rangle \in \overline{D} \times \overline{F}$, so that e.g. the number of firing-places f occurring in a multiset $\overline{f}$ coincides with the marking in the firing-place f—for instance marking $f_1||d_1||d_1||d_2$ means one token in f_1 and d_2, and two tokens in d_1.

Pictorially, a ReSpecT net can be represented in the style of Petri nets: data-places are round nodes, firing-places are square nodes, transitions, incoming arcs, outgoing arcs and inhibitor arcs are denoted as in Petri nets, and firing-places are linked in a one-to-one way to transitions. An example of ReSpecT net is depicted in Fig. 1, where f and f_o are firing-places, the transition is linked to f, d_i and d_r are sources of an incoming arc, d_n is source of an inhibitor arc, d_o, d_r and f_o are targets of outgoing arcs, f has two tokens, f_0 and d_n one, and the other places none.

The semantics of a ReSpecT net is defined as usual by sequences of transitions of markings. Formally, this can be expressed through a transition system $(\overline{D} \times \overline{F}, \longrightarrow_{\mathcal{RST}})$, where notation $\langle \overline{d}, \overline{f} \rangle \longrightarrow_{\mathcal{RST}} \langle \overline{d}', \overline{f}' \rangle$ is used to state that in marking $\langle \overline{d}, \overline{f} \rangle$ a transition can fire which leads to the marking $\langle \overline{d}', \overline{f}' \rangle$. The details of such transitions can be expressed using the following SOS-like (Structural Operational Semantics [20]) rules:

$$\frac{Inh(f) \cap \overline{d} = \oslash \qquad Out(f) = \overline{d}'||\overline{f}'}{\langle In(f)||\overline{d}, f||\overline{f} \rangle \longrightarrow_{\mathcal{RST}} \langle \overline{d}'||\overline{d}, \overline{f}'||\overline{f} \rangle} \quad \text{[SUCC]}$$

$$\frac{d \in Inh(f)}{\langle d||\overline{d}, f||\overline{f} \rangle \longrightarrow_{\mathcal{RST}} \langle d||\overline{d}, \overline{f} \rangle} \quad \text{[FAIL-ABS]}$$

$$\frac{In(f) \not\subseteq \overline{d}}{\langle \overline{d}, f||\overline{f} \rangle \longrightarrow_{\mathcal{RST}} \langle \overline{d}, \overline{f} \rangle} \quad \text{[FAIL-PRES]}$$

A transition is enabled when its firing-place, say it is f, has (at least) one token, in which case at least one of the three rules applies:

- If *(i)* all the sources of incoming arcs $In(f)$ have one token and *(ii)* no source of inhibitor arcs $Inh(f)$ has tokens, then rule [SUCC] applies. As a result, one token from f and from each data-place in $In(f)$ is removed, and one token from each data- and firing-place in $Out(f)$ is added.
- If one of the sources of inhibitor arcs have a token ($d \in Inh(f)$), rule [FAIL-ABS] applies, which simply causes one token from f to be removed.
- If one of the sources of incoming arcs have no tokens ($In(f) \not\subseteq \overline{d}$), rule [FAIL-PRES] applies, which simply causes one token from f to be removed—and similarly if d has more arcs toward f.

A ReSpecT net behaviour can be described in terms of pending reactive computations (also called *reactions*), represented by tokens in firing-places. Each such computation can execute with success ([SUCC]), in which case it removes tokens from sources of incoming arcs $In(f)$ and adds tokens to targets of outgoing arcs $Out(f)$. Vice versa, it could also fail due to the presence of a token in $Inh(f)$ ([FAIL-ABS]) or the absence of a required token in $In(f)$ ([FAIL-PRES]). In all these three cases, a token from f is removed, modelling the fact that the reactive computation has occurred.

2.2 Rule-based Notation

Since a firing-place f is associated to a transition, it also identifies *(i)* the sources of inhibitor arcs $\overline{d}_n = Inh(f)$, *(ii)* the sources of incoming arcs $\overline{d}_i = Inh(f)$, *(iii)* the data-places that are target of outgoing arcs $\overline{d}_o$, and *(iv)* the firing-places that are target of outgoing arcs $\overline{f}_o$ ($\overline{d}_o || \overline{f}_o = Out(f)$). Correspondingly, dually to the graphic notation, a ReSpecT net can be represented by a set of rules—resembling rewrite rules for the dataspace. For each firing-place $f \in F$, we write

$$(\neg \overline{d}_n)[f]\overline{d}_i \longrightarrow_{\mathcal{RST}} \overline{d}_o[\overline{f}_o]$$

There, $\overline{d}_n$ can be thought of as the multiset of dataspace elements that should be absent for the reaction to be successfully executed, $\overline{d}_i$ as the multiset of dataspace elements that should be present for the reaction to be successfully executed, $\overline{d}_o$ as the multiset of dataspace elements inserted when the reaction is executed, f is the firing-elements that should occur—and which enable the reaction—, and $\overline{f}_o$ are the firing-elements inserted after the reaction has been executed ([SUCC]). If at a given time a firing-element occurs but the above conditions on $\overline{d}_n$ and $\overline{d}_i$ are not satisfied, that firing-element disappears modelling the reaction failure ([FAIL-ABS],[FAIL-PRES]). For instance, the ReSpecT net of Fig. 1, which is composed by one transition only, is expressed by the rule:

$$(\neg \mathtt{d_n})[\mathtt{f}]\mathtt{d_i}||\mathtt{d_r} \longrightarrow_{\mathcal{RST}} \mathtt{d_o}||\mathtt{d_r}[\mathtt{f_o}]$$

$$
\begin{array}{lll}
P & \triangleq D \cup F & \text{data- and firing-places} \\
T & \triangleq \{(f, \surd)\} & \text{success (S) transitions} \\
 & \cup \{(f, \neg, d) : d \in In(f)\} & \text{presence-failure (PF) transitions} \\
 & \cup \{(f, \times, d) : d \in Inh f\} & \text{absence-failure (AF) transitions} \\
In_{PT} & \triangleq \{(f, \surd) \mapsto f || In(f)\} & \text{incoming arcs to S transitions} \\
 & \cup \{(f, \neg, d) \mapsto f\} & \text{incoming arcs to PF transitions} \\
 & \cup \{(f, \times, d) \mapsto f || d\} & \text{incoming arcs to AF transitions} \\
Inh_{PT} & \triangleq \{(f, \surd) \mapsto Inh(f)\} & \text{inhibitor arcs to S transitions} \\
 & \cup \{(f, \neg, d) \mapsto d\} & \text{inhibitor arcs to PF transitions} \\
Out_{PT} & \triangleq \{(f, \surd) \mapsto Out(f)\} & \text{outgoing arcs to S transitions} \\
 & \cup \{(f, \times, d) \mapsto d\} & \text{outgoing arcs to AF transitions}
\end{array}
$$

Fig. 2. Mapping from the ReSpecT net $\langle D, F, In, Out, Inh\rangle$ to the Petri net $\langle P, T, In_{PT}, Out_{PT}, Inh_{PT}\rangle$

We often use the rule-based notation instead of the graphical one—even though they are equivalent—to simplify the discussion of properties.

2.3 Mapping ReSpecT *nets to Petri nets*

Petri nets with inhibitor arcs are sufficiently expressive to model the behaviour of ReSpecT nets—that is, any ReSpecT net can be mapped over a Petri net with the same semantics.

Recall that a Petri net (with inhibitor arcs) is a structure $\langle P, T, In_{PT}, Out_{PT}, Inh_{PT}\rangle$, where P is a set of places, T is a set of transitions, $In_{PT} \subseteq T \mapsto \overline{P}$ maps transitions to sources of incoming arcs, $Inh_{PT} \subseteq T \mapsto \overline{P}$ maps transitions to sources of inhibitor arcs, $Out_{PT} \subseteq T \mapsto \overline{P}$ maps transitions to targets of outgoing arcs. Semantics is expressed by the SOS-like rule:

$$\frac{\overline{p} \cap Inh_{PT}(t) = \oslash}{In_{PT}(t)||\overline{p} \longrightarrow_{\mathcal{P}} Out_{PT}(t)||\overline{p}}$$

Then, given a ReSpecT net $\langle D, F, In, Out, Inh\rangle$, the corresponding Petri net $\langle P, T, In_{PT}, Out_{PT}, Inh_{PT}\rangle$ is obtained as shown in Fig. 2. First, we have one place for each data- and firing-place (and the number of tokens in them is unchanged by the mapping). Then, each firing-place f (i.e., each transition in the ReSpecT net) generates three different kinds of transitions: *(i)* a success (S) transition (labelled with $\surd$); *(ii)* one presence-failure (PF) transition (labelled with $\neg$) for each incoming arc to f, modelling failure when checking the presence of tokens in its source; and *(iii)* one absence-failure (AF) transition (labelled with $\times$) for each inhibitor

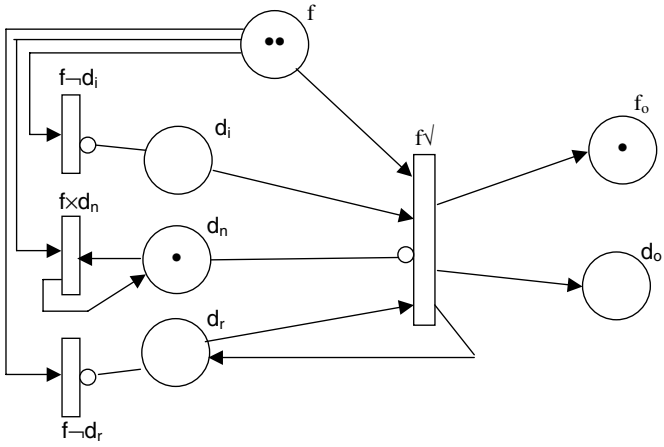

Fig. 3. Petri net for the ReSpecT net of Fig. 1

arc to f, modelling failure when checking the absence of tokens in its source. Each of these three kinds of transitions respectively realises the behaviour resulting from operational rules [SUCC], [ABS], and [PRES] described above.

As an example, the ReSpecT net shown in Fig. 1 turns into the Petri net in Fig. 3.

This mapping preserves operational semantics, in that any transition of a ReSpecT net corresponds to exactly one transition in the Petri net, and vice versa. More precisely, let $|.|^P \subseteq (\overline{D} \times \overline{F}) \mapsto \overline{P}$ be the mapping from ReSpecT nets markings to Petri nets markings, and $\longrightarrow_{\mathcal{P}} \subseteq \overline{P} \times \overline{P}$ the Petri net transition relation shown above, we have:

$$\langle \overline{d}, \overline{f} \rangle \longrightarrow_{\mathcal{RST}} \langle \overline{d}', \overline{f}' \rangle \quad \Leftrightarrow \quad |\langle \overline{d}, \overline{f} \rangle|^P \longrightarrow_{\mathcal{P}} |\langle \overline{d}', \overline{f}' \rangle|^P$$

This result allows us to analyse the properties of ReSpecT nets (and simulate runtime behaviours) reusing results and tools used for Petri nets with inhibitor arcs.

3 The ReSpecT specification language

3.1 ReSpecT *in a nutshell*

ReSpecT [16] is a logic-based language to program the reactive behaviour of tuple centres [17].

Tuple centres are *coordination media* extending the basic model of LINDA tuple spaces [13]. Similarly to LINDA, they accept and serve requests for inserting a tuple `t` (by primitive `out(t)`), removing a tuple matching template `tt` (by primitive `in(tt)`), and reading a tuple matching template `tt` (by primitive `rd(t)`).[1] Differently from LINDA tuple spaces, tuple centres can be programmed so that whenever an external communication event occurs a computation reactively starts which may

[1] Tuple centres can also deal with usual predicative primitives `inp(tt)` and `rdp(tt)` of LINDA, but these are not considered here for the sake of simplicity and without loss of generality.

σ	::= {reaction(p(t),($body$)).}	specification
p	::= cp | rp	ReSpecT primitives
cp	::= out | in | rd	communication primitives
rp	::= in_r | rd_r | out_r | no_r	reaction primitives
$body$	::= [$goal${,$goal$}]	specification body
ph	::= pre | post	direction predicates
$goal$	::= ph | rp(t)	goals

Fig. 4. The core syntax of a ReSpecT specification

affect the state of the inner tuple space. In particular, ReSpecT tuple centres adopt logic tuples (Prolog-like terms with variables) for both tuples and tuple templates, and unification as the matching criterion. External communication events can either be *(i)* a *listening*—the reception of a request from a coordinated process (either a `in`, `rd`, `out`)—, or *(ii)* a *speaking*—the production of a reply towards a coordinated process (the reply to either a `in` or `rd`).

The ReSpecT language can be used to declare a set σ of *reaction specification tuples* (RSTs), using the syntax of Fig. 4. We suppose that $\mathtt{t} \in \mathtt{T}$ ranges over tuples, θ over substitutions of variable to terms, and denote $\theta\mathtt{t}$ the tuple obtained by applying θ to `t`.

Each RST has a head and a body. When a communication event p(t) occurs, all the RSTs with a matching head are activated—that is, each of their bodies—specifying an atomic computation over the tuple centre—is used to spawn a pending reaction waiting to be executed. Reactions are composed by a sequence of reaction primitives rp resembling LINDA primitives, which are used to remove a tuple (`in_r`), read a tuple (`rd_r`), insert a tuple (`out_r`), and check for the absence of a tuple (`no_r`). This sequence can actually contain a direction predicate ph, `pre` or `post`, which is used to filter between reactions to a listening or a speaking. [2]

A reaction is non-deterministically picked and executed, by atomically executing all its reaction primitives. Their effect is to change the state of the tuple centre, and to fire new reactions, as long as they match some other RST—whose head can specify a reaction primitive (internal communication events) other than a communication primitive (external communication events). This recursive creation of reactions is the mechanism by which ReSpecT achieves expressiveness up to reaching Turing-completeness [8].

Primitives `in_r`, `rd_r`, and `no_r` might fail (the former two when the tuple is absent, the latter when it is present), in which case the reaction execution fails, and its effect on the tuple centre is rolled back. The computation fired by an external communication event stops when (if) no more reactions are pending: then, the tuple

[2] Other sorts of ReSpecT primitives actually exist [17], such as logic-based and arithmetic primitives, which however are of little interest here, and so ignored.

```
reaction( out(chops(C1, C2)),
          (in_r(chops(C1, C2)), out_r(chop(C1)), out_r(chop(C2)))).
reaction( in(chops(C1, C2)),
          (pre, out_r(required(C1, C2)))).
reaction( out_r(required(C1, C2)),
          (in_r(chop(C1)), in_r(chop(C2)), out_r(chops(C1, C2)))).
reaction( in(chops(C1, C2)),
          (post, in_r(required(C1, C2)))).
reaction( out_r(chop(C1)),
          (rd_r(required(C1, C), in_r(chop(C1)),
           in_r(chop(C)), out_r(chops(C1,C2))))).
reaction( out_r(chop(C2)),
          (rd_r(required(C, C2), in_r(chop(C)),
           in_r(chop(C2)), out_r(chops(C,C2))))).
```

Fig. 5. ReSpecT specification for the dining philosophers

centre waits until the next communication event occurs.

3.2 An example

A classical example of a ReSpecT specification, taken from [16], is used to deal with the dining philosophers problem [9]—also referred to as hurried philosophers. In its general setting, this example provides a fixed number n of resources, each accessible when two different locks have been acquired, with each of the n locks being shared by two (adjacent) resources. A figurative description of the problem is obtained by considering n (eastern) philosophers willing to eat from n spaghetti dishes in a circular table, but with only one chopstick in between each couple of dishes: a philosopher has to wait for both chopsticks to be available in order to eat.

A Linda tuple space could be exploited to share resources and control accesses, modelling each chopstick as a tuple `chop(C)` to be removed from and reinserted in the tuple space. However, in this case a deadlock situation can occur, as all the philosophers might have one chopstick (e.g., their left one) and indefinitely wait for the other (e.g., their right one). So, the idea is to exploit a tuple centre to allow agents to request a couple of chopsticks atomically: so, while chopsticks are represented through individual tuples of the form `chop(C)`, philosophers first require and then return pairs of tuples (`in_r(chops(C1,C2))` and `out_r(chops(C1,C2))`, respectively). The different philosophers' and the tuple centre representation are bridged by the behaviour of the tuple centre, that reacts to `in_r(chops(C1,C2))` and `out_r(chops(C1,C2))` invocations according to the ReSpecT specification reported in Fig. 5.

Generally speaking, each specification tuple `reaction(`e`,(`g_1`,..,`g_n`))` denotes reaction `(`g_1`,..,`g_n`)` to be executed when a given communication event e occurs, expressed as a sequence of goals $\widehat{g} = g_1, .., g_n$—the void sequence denoted with $\oslash$. Communication events can either be caused by external events (listening or speaking events), or be raised when a reaction primitive is executed. The execution of a reaction is to be considered atomic, in the sense that if any of its goals cannot be executed the whole reaction fails and does not affect the tuple space state in any way.

In the above specification, the first rule is used to convert the release of a couple of chopsticks (`out(chops(C1,C2))`) into two separate releases (`out_r(chop(C1))`

and `out_r(chop(C2))`). When a request for a pair of chopsticks is received, a tuple `required(C1,C2)` is reified in the space by the second rule, which causes, by third rule, the attempt to consume the two single chopsticks, and join them together to make the previous request `in_r(chops(C1,C2))` satisfiable. If this is the case, the fourth rule drops tuple `required(C1,C2)`. Finally, the fifth and sixth rule are used to look for pending requests each time a new single chopstick is released.

3.3 Operational Semantics

We provide the operational semantics of the ReSpecT language by characterising the possible computations fired as an external communication event $e \in E$ occurs. In order to simplify our treatment without loss of generality, we abstract away from the fact that computations can be fired by either a listening or a speaking: the computations fired by the two kinds of events rely on the same model and could be seen as generated by two disjoint sets of RSTs. As a result, we can avoid considering direction predicates (a goal g is then only of the kind $rp(\mathtt{t})$), and e can be considered simply of the kind $cp(\mathtt{t})$.

Then, operational semantics can be formalised in terms of the transition system $\mathcal{R} = \langle \overline{\mathtt{T}}, \longrightarrow_{\mathcal{R}}, E \rangle$, where $\overline{\mathtt{t}} \xrightarrow{e}_{\mathcal{R}} \overline{\mathtt{t}}'$ means that the tuple-multiset $\overline{\mathtt{t}}$ (in the tuple centre) moves to $\overline{\mathtt{t}}'$ due to the computation fired by communication event e.

Reactions are associated to events by a function ρ, which is assumed to take the operation $p(\mathtt{t})$ executed, and yield the reactions that should be correspondingly fired. A reaction r is a sequence of goals $\widehat{g} = g_1; ..; g_n$ each of the kind $rp(\mathtt{t})$, hence we write $\rho_\sigma(p(\mathtt{t})) = \overline{r}$ to say that in the ReSpecT specification σ, the execution of operation $p(\mathtt{t})$ causes the multiset of pending reactions $\overline{r}$ to be created.

The transition relation in $\mathcal{R}$ is now modelled in terms of a completed and (possibly) finite sequence of reaction executions

$$\frac{\langle \overline{\mathtt{t}}, \rho_\sigma(e) \rangle \longrightarrow^*_{\mathcal{E}} \langle \overline{\mathtt{t}}', \oslash \rangle}{\overline{\mathtt{t}} \xrightarrow{e}_{\mathcal{R}} \overline{\mathtt{t}}'}$$

Notice that the sequence of reaction executions might not complete—which can happen because of the Turing-completeness of ReSpecT [8].

The semantics of reaction execution is itself modelled by a transition relation $\longrightarrow_{\mathcal{E}} \subseteq (\overline{\mathtt{T}} \times \overline{R}) \times (\overline{\mathtt{T}} \times \overline{R})$. In each transition $\longrightarrow_{\mathcal{E}}$, a pending reaction is selected, and its goals sequentially executed until completion (the sequence of goals r becomes the void sequence of goals $\oslash$)—in which case the effects to tuples and pending reactions are applied—or until some goal cannot be executed ($\nrightarrow_{\mathcal{G}}$)— in which case

such effects are discarded.

$$\frac{\langle \overline{\mathtt{t}}, \oslash, r\rangle \longrightarrow_{\mathcal{G}}^{*} \langle \overline{\mathtt{t}}', \overline{r}', \oslash\rangle}{\langle \overline{\mathtt{t}}, r||\overline{r}\rangle \longrightarrow_{\mathcal{E}} \langle \overline{\mathtt{t}}', \overline{r}||\overline{r}'\rangle} \qquad \text{[SUCCESS]}$$

$$\frac{\langle \overline{\mathtt{t}}, \oslash, r\rangle \longrightarrow_{\mathcal{G}}^{*} \langle \overline{\mathtt{t}}', \overline{r}', g; \widehat{g}\rangle \not\rightarrow_{\mathcal{G}}}{\langle \overline{\mathtt{t}}, r||\overline{r}\rangle \longrightarrow_{\mathcal{E}} \langle \overline{\mathtt{t}}, \overline{r}\rangle} \qquad \text{[FAILURE]}$$

Finally, the transition relation $\longrightarrow_{\mathcal{G}}$ described by the following rules defines the semantics of goal execution.

$$\langle \overline{\mathtt{t}}, \overline{r}, \mathtt{out\_r(t)}; \widehat{g}\rangle \longrightarrow_{\mathcal{G}} \langle \mathtt{t}||\overline{\mathtt{t}}, \overline{r}||\rho(\mathtt{out\_r(t)}), \widehat{g}\rangle$$

$$\langle \overline{\mathtt{t}}||\theta\mathtt{t}, \overline{r}, \mathtt{rd\_r(t)}; \widehat{g}\rangle \longrightarrow_{\mathcal{G}} \langle \overline{\mathtt{t}}||\theta\mathtt{t}, \overline{r}||\rho(\mathtt{rd\_r(t)}), \theta\widehat{g}\rangle$$

$$\langle \overline{\mathtt{t}}||\theta\mathtt{t}, \overline{r}, \mathtt{in\_r(t)}; \widehat{g}\rangle \longrightarrow_{\mathcal{G}} \langle \overline{\mathtt{t}}, \overline{r}||\rho(\mathtt{in\_r(t)}), \theta\widehat{g}\rangle$$

$$\langle \overline{\mathtt{t}}, \overline{r}, \mathtt{no\_r(t)}; \widehat{g}\rangle \longrightarrow_{\mathcal{G}} \langle \overline{\mathtt{t}}, \overline{r}||\rho(\mathtt{no\_r(t)}, \widehat{g}\rangle \qquad \text{if } \theta t \notin \overline{t}$$

Executing a reaction primitive causes new pending reactions to be fired (e.g. $\rho(\mathtt{no\_r(t)})$), as well as the state of the tuple centre to be affected: `out_r` inserting the tuple, `rd_r` checking for the presence of a matching tuple, `in_r` removing a matching tuple, and `no_r` checking for the absence of a matching tuple. Notice also that in the case primitives `rd_r` and `in_r` involve a substitution, namely tuple θ`t` occurs when `t` is required, the substitution θ is applied on the body continuation $\widehat{g}$.

4 From ReSpecT specifications to ReSpecT nets

As far as analysis is concerned, in this paper we address only ReSpecT specifications that do not contain variables—that is, we focus on *ground* ReSpecT specifications. Whereas this simplification seems to considerably reduce expressiveness, still it is able to model a number of interesting scenarios. On the one hand, this kind of specifications still can model coordination laws in control-oriented scenarios where the "content" of interaction messages can be abstracted away—such as e.g. in most workflow applications [22]. On the other hand, it allows us to describe those applications where the set of tuples used ranges over a finite set D, so that executing e.g. `in(tt)` means to execute either of `in(t_d)` where `t_d` ranges in D. Moreover, we should also notice that most of the work developed so far in the context of analysis of coordination models—see e.g. Busi et al.'s [4] and subsequent works—makes the same assumption. We recognise the need for overcoming this difficulty, but we also believe that the work presented here is a necessary intermediate step to evaluate the applicability of standard models such as Petri nets to the context of the ReSpecT language. The non-trivial extension to full ReSpecT specifications is left as future

work.

A further condition on the structuring of RSTs is actually required, which is called *one-testing*—described in Section 4.3.4—and which leads to the fragment of ReSpecT we analyse here, called $\mathsf{ReSpecT}_g^{1t}$. Any specification in this fragment can be turned into a ReSpecT net modelling the same set of computations. The basic idea of mapping ReSpecT specifications to ReSpecT nets is that any RST would correspond to a firing-place (and to its corresponding transition), any pending reaction waiting to be executed to a token in a firing-place, any tuple occurring in the tuple centre to a token inside a data-place. To show the details of this mapping we proceed in two steps: we first study the mapping for a particular kind of ReSpecT specifications, which we call *flow-oriented*, and then study how any $\mathsf{ReSpecT}_g^{1t}$ specification can be turned into a flow-oriented one.

4.1 Flow-oriented specifications

We introduce the concept of flow-oriented ReSpecT specification. This is a specification structured so that operations affecting data are clearly separated from operations dealing with flow control: in particular, this allows us to emphasise the computation flow that is fired as an external communication event occurs.

A ReSpecT specification is called flow-oriented if its RSTs are of any of the two kinds

```
reaction(cp(t_e)),(        handling a communication primitive (c-RST)
    rd_r(t_f_e)
)).

reaction(rd_r(t_f),(           handling a reaction primitive (r-RST)
    no_r(t_d_n^1),no_r(t_d_n^2),..,     checking the absence of some data
    rd_r(t_d_r^1),rd_r(t_d_r^2),..,     checking the presence of some data
    in_r(t_d_i^1),in_r(t_d_i^2),..,     removing some data
    out_r(t_d_o^1),out_r(t_d_o^2),..,   inserting some data
    rd_r(t_f_1),rd_r(t_f_2),..          firing other reactions
)).
```

where tuples of the kind `t_f` bijects with firing-places $f \in F$, and tuples `t_d` (disjoint from tuples `t_f`) bijects with data-places $d \in D$. We call RSTs of the first kind *c-RST* (RST for communications), and those of the second kind *r-RST* (RST for reactions). Moreover, we suppose that c-RST have different tuples $\mathtt{t\_f_e}$, and similarly, that r-RST have different tuples `t_f`.

4.2 Mapping flow-oriented specifications

To obtain a ReSpecT net from a flow-oriented specification, other than sets D and F seen above, we should specify structures In, Out, Inh, which are easily described in terms of rules $\longrightarrow_{\mathcal{RST}}$. On the one hand, for each c-RST seen above, we add one rule to the ReSpecT net, which accounts for the effect of primitive $cp(\texttt{t\_e})$, and invokes the reaction corresponding to $\texttt{rd\_r}(\texttt{t\_f}_\texttt{e})$. Depending on the communication event we have the following:

Communication event	Phase	ReSpecT net rule
$\texttt{rd\_r}(\texttt{t\_f}_\texttt{e})$	`pre`	$(\neg\oslash)[\oslash]f'_e \longrightarrow_{\mathcal{RST}} \oslash[f_e]$
$\texttt{rd\_r}(\texttt{t\_f}_\texttt{e})$	`post`	$(\neg\oslash)[d_e]f'_e \longrightarrow_{\mathcal{RST}} d_e[f_e]$
$\texttt{in\_r}(\texttt{t\_f}_\texttt{e})$	`pre`	$(\neg\oslash)[\oslash]f'_e \longrightarrow_{\mathcal{RST}} \oslash[f_e]$
$\texttt{in\_r}(\texttt{t\_f}_\texttt{e})$	`post`	$(\neg\oslash)[d_e]f'_e \longrightarrow_{\mathcal{RST}} \oslash[f_e]$
$\texttt{out\_r}(\texttt{t\_f}_\texttt{e})$	`pre`	$(\neg\oslash)[\oslash]f'_e \longrightarrow_{\mathcal{RST}} d_e[f_e]$

On the other hand, for each of the r-RST above we add to the ReSpecT net the rule:

$$(\neg d_n^1||d_n^2||..)[f](d_r^1||d_r^2||..||d_i^1||d_i^2||..) \longrightarrow_{\mathcal{RST}} (d_r^1||d_r^2||..||d_o^1||d_o^2||..)[g_1||g_2||..]$$

Due to the semantics of primitive `no_r` (checking for absence), `rd_r` (checking for presence), `in_r` (removing), and `out_r` (inserting), one easily recognises the correspondence between specification and net.

We are interested in simulating the computation resulting from the occurrence of a single event $cp(\texttt{t\_e})$, denoting by $\overline{\texttt{t\_d}_0}$ the tuples occurring in the tuple centre at that time. As the initial state for the tuple centre, we consider the one obtained by including one copy of the tuple `t_f` for each RST for reactions: notice that such tuples are neither removed nor added during computation. Therefore we consider as the initial marking of the ReSpecT net the configuration $\langle \overline{d}_0, f_e \rangle$, where tuples correspond to tokens in data-places, and only one token occurs in the firing-place f_e. The following result holds:

$$\overline{\texttt{t\_d}}_0 \xrightarrow{e}_{\mathcal{R}} \overline{\texttt{t\_d}} \quad \Leftrightarrow \quad \langle \overline{d}_0, f_e \rangle \longrightarrow^*_{\mathcal{RST}} \langle \overline{d}, \oslash \rangle$$

That is, any ReSpecT computation fired as the communication event e occurs is mimicked by a corresponding completed evolution $\langle \overline{d}_0, f_e \rangle \longrightarrow^*_{\mathcal{RST}} \langle \overline{d}, \oslash \rangle$ of the ReSpecT net, and vice versa.

Notice that this structuring of ReSpecT specifications amounts at making them "more imperative" in a sense, stressing the difference between primitives computing over data and primitives affecting flow control. This is why we sometime refer to "invoking a RST" when a `rd_r(t_f)` is executed, as it actually causes the reaction

with head `rd_r(t_f)` to be created, and then wait for execution. Whereas the structuring imposed to flow-oriented specifications might not be a good idiom for ReSpecT programming, it is useful to fill the gap between the logic-based language ReSpecT and the control-oriented formalism of ReSpecT nets and Petri nets. In fact, we show here that any $\mathsf{ReSpecT}_g^{1t}$ specification can be automatically turned into a flow-oriented ReSpecT specification—e.g. by a verifier—from which an equivalent Petri net can then be derived.

4.3 Making a ReSpecT *specification flow-oriented*

To make a ground ReSpecT specification flow-oriented, we proceed in the following steps: *(i)* translate c-RSTs, *(ii)* translate r-RSTs and their calls, *(iii)* resolve head clashing in different r-RSTs, and *(iv)* reorder the body of r-RSTs. These four steps are described in detail in the following. For shortness, we name here specification tuples `re(head,body)` instead of `reaction(head,body)`.

4.3.1 Translating c-RSTs

Consider a general RST intercepting a communication primitive e (which is of the kind $cp(\mathtt{t})$), and then executing a sequence of goals. This is translated into two RSTs: the former is a r-RST used to execute that sequence of goals, the latter is a c-RST that intercepts e and invokes the former. In the case where more RSTs exist that intercept e, only one c-RST has to be generated, which should actually invoke all the r-RSTs created. Formally:

$$\begin{array}{lcl} \mathtt{re}(e,(\widehat{g}_1)). & \mapsto & \mathtt{re}(e,(\mathtt{rd\_r}(\mathtt{t\_f}_\mathtt{e}^\mathtt{1}),..,\mathtt{rd\_r}(\mathtt{t\_f}_\mathtt{e}^\mathtt{n}))). \\ .. & & \mathtt{re}(\mathtt{rd\_r}(\mathtt{t\_f}_\mathtt{e}^\mathtt{1}),(\widehat{g}_1)). \\ \mathtt{re}(e,(\widehat{g}_n)). & & .. \\ & & \mathtt{re}(\mathtt{rd\_r}(\mathtt{t\_f}_\mathtt{e}^\mathtt{n}),(\widehat{g}_n)). \end{array}$$

where $\mathtt{t\_f}_\mathtt{e}^\mathtt{1},..,\mathtt{t\_f}_\mathtt{e}^\mathtt{n}$ are new (and different) tuples. By this initial step, all c-RSTs have been created, and only r-RSTs have then to be considered from here on.

4.3.2 Translating r-RSTs

As a second step we deal with interception of reaction primitives. First, the head of each r-RST is translated into the form $\mathtt{rd\_r}(\mathtt{t\_f})$, and any invocation to it is correspondingly accommodated. Formally:

$$\begin{array}{lcl} \mathtt{re}(rp(\mathtt{t\_d}),(\widehat{g})). & \mapsto & \mathtt{re}(\mathtt{rd\_r}(\mathtt{t\_f}_\mathtt{d}^{rp}),(\widehat{g})). \\ \mathtt{re}(p(\mathtt{t}),(\widehat{g};rp(\mathtt{t\_d});\widehat{g}')). & \mapsto & \mathtt{re}(p(\mathtt{t}),(\widehat{g};rp(\mathtt{t\_d});\mathtt{rd\_r}(\mathtt{t\_f}_\mathtt{d}^{rp});\widehat{g}')). \end{array}$$

4.3.3 Resolving head clashing

In the translation executed so far, it may still happen that two r-RSTs have the same head. To prevent this, we rename clashing heads and accommodate invocations so

first \ second	no_r	rd_r	in_r	out_r
no_r	no_r	*fail*	*fail*	*unchanged*
rd_r	*fail*	rd_r	in_r	*unchanged*
in_r	*unchanged*	*unchanged*	*unchanged*	rd_r
out_r	*fail*	out_r	*drop*	*unchanged*

Fig. 6. Local translation of reaction bodies

that *all* the matching r-RSTs are concurrently invoked. Formally:

$$\begin{array}{lcl} \mathtt{re}(\mathtt{rd\_r}(\mathtt{t\_f}_{\mathtt{d}}^{rp}), (\widehat{g}_1)). & \mapsto & \mathtt{re}(\mathtt{rd\_r}(\mathtt{t\_f}_{\mathtt{d,1}}^{rp}), (\widehat{g}_1)). \\ .. & & .. \\ \mathtt{re}(\mathtt{rd\_r}(\mathtt{t\_f}_{\mathtt{d}}^{rp}), (\widehat{g}_n)). & & \mathtt{re}(\mathtt{rd\_r}(\mathtt{t\_f}_{\mathtt{d,n}}^{rp}), (\widehat{g}_n)). \\ \\ \mathtt{re}(.., (\widehat{g}; \mathtt{rd\_r}(\mathtt{t\_f}_{\mathtt{d}}^{rp}); \widehat{g}')). & \mapsto & \mathtt{re}(.., (\widehat{g}; \mathtt{rd\_r}(\mathtt{t\_f}_{\mathtt{d,1}}^{rp}); ..; \mathtt{rd\_r}(\mathtt{t\_f}_{\mathtt{d,n}}^{rp}); \widehat{g}')). \end{array}$$

4.3.4 Reordering bodies

Bodies of r-RSTs are now formed by reactive primitives applied to tuples `t_d`, along with goals of the kind `rd_r(t_f)`. In order to obtain a sequence of goals adhering to the final structure of r-RSTs, a local translation is to be performed.

If for no tuple `t` there are different primitives working on it, then primitives never interfere with each other, hence suffices it to reorder goals so that their primitives follow the order `no_r`, `rd_r`, `in_r`, `out_r`.

In the opposite case, any subsequence of operations working on the same tuples have to be ordered first, and then all the subsequences can be simply merged—without risk of semantic interference. For instance, from the body

$$\mathtt{out\_r}(\mathtt{t\_d}^1), \mathtt{rd\_r}(\mathtt{t\_d}^2), \mathtt{out\_r}(\mathtt{t\_d}^1), \mathtt{in\_r}(\mathtt{t\_d}^2)$$

the two subsequences $\mathtt{out\_r}(\mathtt{t\_d}^1), \mathtt{out\_r}(\mathtt{t\_d}^1)$ and $\mathtt{rd\_r}(\mathtt{t\_d}^2), \mathtt{in\_r}(\mathtt{t\_d}^2)$ can be extracted: the former is correctly ordered, the second is equivalent to the single primitive $\mathtt{in\_r}(\mathtt{t\_d}^2)$. Hence, the whole body can be rewritten by merging the two obtaining:

$$\mathtt{in\_r}(\mathtt{t\_d}^2), \mathtt{out\_r}(\mathtt{t\_d}^1), \mathtt{out\_r}(\mathtt{t\_d}^1)$$

In particular, to order one subsequence formed by reaction primitives over the same tuple `t`, one should take any subsequent couple of goals in it, translate such a couple as depicted in Fig. 6, and iterate this process until reaching a fixpoint.

Rows range over the first element of the couple, and columns over the second. Cells content is as follows: *fail* refers to couples that surely make the reaction fail, hence the whole body can be simply left void; *unchanged* means that the couple is to be left as it is; *drop* means that the couple is to be dropped from the body; a single reaction primitive (`out_r`, `rd_r` or `in_r`) means that the primitive should

substitute the corresponding couple. It is easy to recognise that any change to a couple never alter the semantics of the whole body.

The couple `in_r(t), no_r(t)` is the only unchanged one that is out of order according to the flow-oriented structuring. This implies that by applying the translation of couples until reaching the fixpoint we obtain sequences which either satisfy the r-RST ordering or begin with a structure of the kind below:

$$\texttt{in\_r(t)}, \texttt{in\_r(t)}, .., \texttt{in\_r(t)}, \texttt{no\_r(t)}$$

The occurrence of one such sequence amounts to test whether exactly n copies of `t` reside in the space—where n is the number of `in_r(t)` operations.

To the best of our knowledge, no extension of Petri nets studied so far deals with this case: e.g. Petri nets with weighted inhibitor arcs, allows for testing whether *at most* n tuples occur, and Contextual Nets [15] whether *at least* n tuples occur—which are significantly different behaviours. Therefore, we opt for leaving this case out of our analysis methodology—studying properties of these kinds of net is interesting and could be subject of our future research.

A specification is called one-testing if this case never happens, so that it can be turned into a flow-oriented specification allowing for an encoding into a Petri net with inhibitory arcs.

5 On the Analysis of ReSpecT nets

The methodology studied in this paper makes it possible to automatically derive a Petri net with inhibitor arcs modelling computations of a given ReSpecT specification. The encoding is also rather direct: the occurrence of tuples and pending reactions is modelled by tokens in specific places, and success/failure of executions of RSTs by specific transitions. This means that all the exiting tools for Petri nets (with inhibitor arcs) can be directly exploited to verify properties—such as safety and liveness—of ReSpecT specifications [14,5]. A particularly relevant safety property in the context of coordination is covering [23], stating that a given submarking will never be reached, which translates into the fact that a given (unsafe) configuration of tuples will never occur in the tuple centre.

The presence of inhibitor arcs makes the formalism of Petri nets Turing-complete [3], thus most interesting properties—including covering—become undecidable: the only viable approach in this case is to model-check a finite portion of the system. This problem can be seen in connection with the framework of well-structured transitions systems [12]: inhibitor arcs make the Petri nets formalism lose its well-structure—in a sense, its monotonic behaviour with respect to operational semantics and inclusion of markings. As common in these situations, sufficient conditions for decidability are still worth investigating [23,3,2]. In particular, the ReSpecT programming practice [8] shows that a number of interesting and useful specifications do conserve an intrinsic well-structure, and might then in principle be analysed by a formal tool for effectively proving properties of interest.

Hence, in this section we provide some results concerning elimination of inhibitor

arcs in ReSpecT nets—similarly in aim to [3,4]—which may be the basis for a full-featured analysis methodology for ReSpecT specifications.

5.1 Weakening the net

We first assume that the ReSpecT specification makes no use of primitive `no_r(t)`. This means that relation *Inh* is empty and, according to the mapping in Fig. 2, no transition $\langle f, \times, d\rangle$ occurs in the net and no transition $\langle f, \surd\rangle$ has inhibitor arcs. The remaining inhibitor arcs are then of the kind $\{(f, \neg, d) \mapsto d\}$, such as e.g. the inhibitor arc from places d_i and d_r in Fig. 3. Their role is to make a reaction fail when it includes goals `rd_r(t)` or `in_r(t)` and the tuple `t` is absent.

To tackle the intricacies introduced in the analysis of such inhibitor arcs, in this section we describe two techniques to simplify the Petri net so that such arcs are dropped without altering the safety properties of the modelled system.

A first, simple approach consists in the trivial withdrawal of all inhibitor arcs to transitions $(f, \neg, d)$: after that the behaviour of those transitions is to simply remove tokens from f as they occur—see Fig. 3. The net we obtain describes the semantics of the original ReSpecT specification under the assumption that as a reaction is pending and waiting for one or more tuples to occur, it can simply disappear—as it would have failed, or as it were never fired at all. That is, a reaction including a `rd_r(t)` or `in_r(t)` could fail even though `t` is present.

It is easy to recognise that the net obtained by this approach, which has same set of places and transitions, allows for a strictly greater set of (completed) computations—those featuring such new dummy failings. In particular, denoted by $\longrightarrow_{\mathcal{A}}$ the operational semantics of the original net and by $\longrightarrow_{\mathcal{B}}$ the one without inhibitor arcs, we have:

$$\langle \overline{d}, f\rangle \longrightarrow^*_{\mathcal{A}} \langle \overline{d}\,', \oslash\rangle \qquad \Rightarrow \qquad \langle \overline{d}, f\rangle \longrightarrow^*_{\mathcal{B}} \langle \overline{d}\,', \oslash\rangle$$

This result is not generally entailed when trivially removing inhibitor arcs from Petri nets [4,3], but holds here for inhibitor arcs are used to simply drop tokens from firing-places, that is, to prevent some reactions to be executed. The importance of this result lies in the fact that if system $\mathcal{B}$ is proved safe—e.g. it does not cover an unsafe marking—then system $\mathcal{A}$ is safe as well. Still, one can argue that $\mathcal{B}$ might include a significantly greater set of behaviours, so that many safety properties of $\mathcal{A}$ are never reflected in $\mathcal{B}$.

A more refined approximation to system $\mathcal{A}$ than system $\mathcal{B}$ can be obtained by dropping from system $\mathcal{A}$ all transitions $\langle f, \neg, d\rangle$ along with their inhibitor and incoming arcs. The resulting system, denoted by $\mathcal{C}$, corresponds to the idea that pending reactions never fail, but simply reside in the system until they can successfully execute or until deadlock. In other words, this approximation amounts to interpret the ReSpecT net as a simple Petri net, considering firing-places as standard Petri net places. The system $\mathcal{C}$ obtained has a smaller set of completed computations than $\mathcal{A}$—hence, in a sense, it does not lead to many behaviours. Still, $\mathcal{C}$ and $\mathcal{A}$ have precisely the same set of (possibly) uncompleted computations over tokens

in data-places d:

$$\langle \overline{d}, \overline{f} \rangle \longrightarrow^*_{\mathcal{A}} \langle \overline{d}', \overline{f}' \rangle \qquad \Leftrightarrow \qquad \langle \overline{d}, \overline{f} \rangle \longrightarrow^*_{\mathcal{C}} \langle \overline{d}', \overline{f}'' \rangle$$

That is, starting from the same marking they are able to reach the same markings $\overline{d}'$ over data-places. As an example consider the ReSpecT net rules:

$$(\neg\oslash)[f_1]\oslash \longrightarrow_{\mathcal{RST}} d_1[\oslash]$$
$$(\neg\oslash)[f_2]d_1 \longrightarrow_{\mathcal{RST}} d_2[\oslash]$$

Starting from marking $\langle \oslash, f_1||f_2 \rangle$, system $\mathcal{A}$ can feature the two completed computations

$$\langle \oslash, f_1||f_2 \rangle \longrightarrow_{\mathcal{A}} \langle \oslash, f_1 \rangle \longrightarrow_{\mathcal{A}} \langle d_1, \oslash \rangle$$
$$\langle \oslash, f_1||f_2 \rangle \longrightarrow_{\mathcal{A}} \langle d_1, f_2 \rangle \longrightarrow_{\mathcal{A}} \langle d_2, \oslash \rangle$$

while system $\mathcal{C}$ only features the latter:

$$\langle \oslash, f_1||f_2 \rangle \longrightarrow_{\mathcal{C}} \langle d_1, f_2 \rangle \longrightarrow_{\mathcal{C}} \langle d_2, \oslash \rangle$$

However both systems can reach the same set of two markings $\{d_1\}$ and $\{d_2\}$. Hence, analogously to the former approach, covering-based safety of system $\mathcal{C}$—which can be proved by an automatic tool—entails safety of $\mathcal{A}$. Still, this second approach provides a rather refined approximation, and is then a better candidate for an analysis methodology for ReSpecT specifications.

5.2 Accelerations

Inhibitor arcs not only model failures, but are also used to model the semantics of primitive `no_r(t)`. In a number of interesting cases, ReSpecT specifications do use this primitive, which is in fact necessary to make the ground version of ReSpecT language Turing-complete.

Often, primitive `no_r(t)` is used to check whether a transformation process over set of tuples is over, though the overall process is still a monotonic one preserving the well-structure. A simple example is the case where *all* the tuples *t_a* occurring in the tuple space have to be substituted by tuples *t_b*. This interaction pattern resembles broadcast protocols [10], which are shown to retain the well-structure property [12]. Such a tuple transformation can e.g. be realised by the flow-oriented ReSpecT specification:

$\texttt{reaction}(\texttt{rd\_r}(\texttt{t\_f}_{\texttt{go}}), \texttt{out}(\texttt{t\_d}), \texttt{rd\_r}(\texttt{t\_f}_{\texttt{next}}), \texttt{rd\_r}(\texttt{t\_f}_{\texttt{last}})).$

$\texttt{reaction}(\texttt{rd\_r}(\texttt{t\_f}_{\texttt{next}}), (\texttt{in\_r}(\texttt{t\_a}), \texttt{out\_r}(\texttt{t\_b}), \texttt{rd\_r}(\texttt{t\_f}_{\texttt{next}}), \texttt{rd\_r}(\texttt{t\_f}_{\texttt{last}}))).$

$\texttt{reaction}(\texttt{rd\_r}(\texttt{t\_f}_{\texttt{last}}), (\texttt{no\_r}(\texttt{t\_a}), \texttt{rd\_r}(\texttt{t\_f}_{\texttt{out}}), \texttt{in\_r}(\texttt{t\_d}))).$

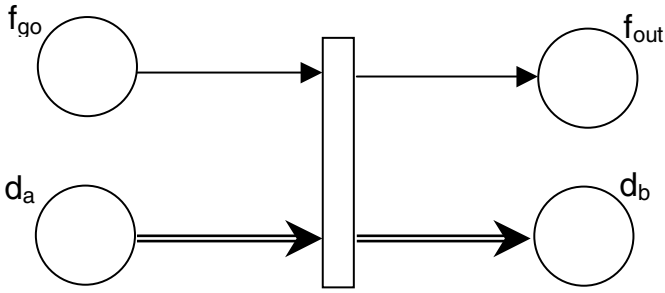

Fig. 7. Petri net with transfer arcs

As the process is started by firing-tuple $\texttt{t\_f}_{\texttt{go}}$, tuple `t_d` is inserted and the two reactions $\texttt{rd\_r}(\texttt{t\_f}_{\texttt{next}})$ and $\texttt{rd\_r}(\texttt{t\_f}_{\texttt{last}})$ are fired: the former substitutes one occurrence of `t_a` with `t_b` and proceeds recursively, the latter *(i)* checks the absence of `t_a`, *(ii)* fires the escaping tuple $\texttt{t\_f}_{\texttt{out}}$, and *(iii)* drops `t_d` to avoid multiple firing of $\texttt{t\_f}_{\texttt{out}}$. These kinds of idiom are quite common in ReSpecT programming, and are the main exploitation of primitive `no_r(t)`.

In the context of program analysis, a technique called *acceleration* has been introduced to simplify the treatment of these situations [21,11,1]. Translated to our setting, it amounts to substitute the Petri net for the above specification to the very simple Petri net of Fig. 7, featuring so-called *transfer arcs*. Transfer arcs are used to make a transition affect *all* the tokens in a place. In this Petri net, when a token appears in f_{go} the transition can occur, which causes the token to be moved to place f_{out} as well as all tokens in d_a to be transferred to d_b.

Petri nets with transfer arcs and without inhibitor arcs are well-structured systems, hence for instance covering is decidable [12]. Therefore, it would be interesting to study analysis methodologies identifying fragments of ReSpecT specifications which allow for accelerations, resulting in the removal of some/all inhibitor arc.

6 Example

In this section we provide details about a simple application of our methodology, based on the philosophers example reported in Section 3.2. As a first step, we consider the ground version of the program, sticking to the case where 3 philosophers coordinate for the acquisition of chops denoted by tuples `c1`, `c2`, and `c3`, through requests specifying tuples `c12`, `c23`, and `c31`—modelling tuples of the kind `chop(C)` and `chops(C1,C2)` respectively, as shown in the code of Fig. 5. Such requests are reified in the tuple centre through tuples `r12`, `r23`, and `r31`—modelling tuples `request(C1,C2)`.

Then, by applying the mapping described in this paper, we obtain the ReSpecT net described by the rules below—each rule should actually appear in the three versions concerning the three resources, the one for the first resource is only reported

here for brevity.

$(\neg\oslash)[f'_{o12}]\oslash \longrightarrow_{\mathcal{RST}} c12[f_{o12}]$ c-RST for `out(c12)`

$(\neg\oslash)[f'_{i12pre}]\oslash \longrightarrow_{\mathcal{RST}} \oslash[f_{i12pre}]$ c-RST for `in(c12):pre`

$(\neg\oslash)[f'_{i12post}]c12 \longrightarrow_{\mathcal{RST}} \oslash[f_{i12post}]$ c-RST for `in(c12):post`

$(\neg\oslash)[f_{o12}]c12 \longrightarrow_{\mathcal{RST}} c1||c2[f^l_{c1}||f^r_{c1}||f^l_{c2}||f^r_{c2}]$ 1st r-RST

$(\neg\oslash)[f_{i12pre}]\oslash \longrightarrow_{\mathcal{RST}} r12[f_{r12}]$ 2nd r-RST

$(\neg\oslash)[f_{r12}]c1||c2 \longrightarrow_{\mathcal{RST}} c12[\oslash]$ 3rd r-RST

$(\neg\oslash)[f_{i12post}]r12 \longrightarrow_{\mathcal{RST}} \oslash[\oslash]$ 4th r-RST

$(\neg\oslash)[f^l_{c1}]r12||c1||c3 \longrightarrow_{\mathcal{RST}} r12||c31[\oslash]$ 5th r-RST

$(\neg\oslash)[f^r_{c1}]r12||c1||c2 \longrightarrow_{\mathcal{RST}} r12||c12[\oslash]$ 6th r-RST

By applying the mapping described in Fig. 2, one can automatically obtain a Petri net describing the behaviour of the specification. Moreover, as the primitive `no_r` is never exploited in the specification, no inhibitory arcs appear in success transitions. Therefore the net can be translated into a simplified Petri net without inhibitory arcs, over which control state reachability properties can be studied.

As a possible tool one can rely on MSR(C) described in [7], where the ReSpecT net could be encoded straightforwardly in the model:

$$
\begin{aligned}
go12 &\longrightarrow c12 \mid fo12\\
gi12pre &\longrightarrow fi12pre\\
gi12post \mid c12 &\longrightarrow gi12post\\
fo12 \mid c12 &\longrightarrow c1 \mid c2 \mid flc1 \mid frc1 \mid flc2 \mid frc2\\
f12pre &\longrightarrow r12 \mid fr12\\
fr12c1 \mid c2 &\longrightarrow c12\\
fi12post \mid r12 &\longrightarrow 0\\
flc1 \mid r12 \mid c1 \mid c3 &\longrightarrow r12 \mid c31\\
frc1 \mid r12 \mid c1 \mid c2 &\longrightarrow r12 \mid c12
\end{aligned}
$$

By using for instance the prototype system described in [6] one could verify whether unsafe configurations are never reached from a given initial state [23]. For instance, consider the initial state $go12 \mid c3$, meaning that a client is releasing locks 1 and 2. Then, it can be proven that the following unsafe configurations are never reached: *(i)* $c1 \mid c1$, two copies of a single lock are never concurrently created; *(ii)* $c12 \mid c12$, two copies of an atomic lock are never concurrently created; *(iii)* $c1 \mid c12$, two copies

of a lock are never concurrently created; and *(iv) r12*, no new request is created.

7 Conclusions and Open Issues

Generally speaking, this article is meant to provide a meaningful example of how formal techniques can be applied to advanced models and infrastructures for the coordination of complex software systems, finding out a suitable compromise between the needs for a high expressiveness of the coordination abstractions, and the limitations imposed by formal frameworks to make relevant properties verifiable. Along this line, ReSpecT nets were introduced to suitably bridge between the ReSpecT logic-based language for the specification of the behaviour of tuple centres, and Petri nets. According to the first relevant results presented here, ReSpecT nets provide the conceptual and technical grounding of a methodology for the analysis of ReSpecT tuple centre behaviour specifications.

The main limit of the approach presented in this paper is the lack of treatment for logic variables and unification. On the one hand, this a typical problem due to the intrinsic dichotomy between operational and declarative, logic-based formal frameworks. On the other hand, a possible solution to this problem is to exploit techniques of *groundisation* of general, non-ground ReSpecT specifications: under simple hypotheses like a finite alphabet and finite data structures (such as non-recursive functors), the number of possible ground versions of any general ReSpecT specification is finite, and the automatic translation techniques introduced in this paper become applicable again. Future work is likely to follow such research line soon.

References

[1] Bardin, S., A. Finkel, J. Leroux and L. Petrucci, *FAST: Fast Acceleration of Symbolic Transition systems*, in: *Computer Aided Verification*, LNCS **2725**, Springer-Verlag, 2003 pp. 118–121.

[2] Bozzano, M. and G. Delzanno, *Beyond parameterized verification*, in: *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS **2280**, Springer-Verlag, 2002 pp. 221–235.

[3] Busi, N., *Analysis issues in Petri nets with inhibitor arcs*, Theoretical Computer Science **275** (2002), pp. 127–177.

[4] Busi, N., R. Gorrieri and G. Zavattaro, *On the expressiveness of Linda coordination primitives*, Information and Computation **156** (2000), pp. 90–121.

[5] *Petri nets tools and software* (2003).
URL http://www.daimi.au.dk/PetriNets/tools/

[6] Delzanno, G., *A constraint-based framework for the automated verification of infinite-state concurrent systems* (2002), prototype.
URL http://www.disi.unige.it/person/DelzannoG/MSR/

[7] Delzanno, G., *An overview of MSR(C): A CLP-based framework for the symbolic verification of parameterized concurrent systems*, Electronic Notes in Theoretical Computer Science **76** (2002).

[8] Denti, E., A. Natali and A. Omicini, *On the expressive power of a language for programming coordination media*, in: *1998 ACM Symposium on Applied Computing (SAC'98)*, Atlanta, GA, USA, 1998, pp. 169–177.

[9] Dijkstra, E., "Co-operating Sequential Processes," Academic Press, London, 1965.

[10] Esparza, J., A. Finkel and R. Mayr, *On the verification of broadcast protocols*, in: *14th Annual IEEE Symposium on Logic in Computer Science (LICS 1999)* (1999), pp. 352–359.

[11] Finkel, A. and J. Leroux, *How to compose Presburger-accelerations: Applications to broadcast protocols*, in: *Foundations of Software Technology and Theoretical Computer Science*, LNCS **2556**, Springer-Verlag, 2002 pp. 145–156.

[12] Finkel, A. and P. Schnoebelen, *Well-structured transition systems everywhere!*, Theoretical Computer Science **256** (2001), pp. 63–92.

[13] Gelernter, D., *Generative communication in Linda*, ACM Transactions on Programming Languages and Systems **7** (1985), pp. 80–112.

[14] Karp, R. and R. Miller, *Parallel programming schemata*, Journal on Computer and System Sciences **3** (1969), pp. 147–195.

[15] Montanari, U. and F. Rossi, *Contextual nets*, Acta Informatica **32** (1995), pp. 545–596.

[16] Omicini, A. and E. Denti, *Formal* ReSpecT, Electronic Notes in Theoretical Computer Science **48** (2001), pp. 179–196.

[17] Omicini, A. and E. Denti, *From tuple spaces to tuple centres*, Science of Computer Programming **41** (2001), pp. 277–294.

[18] Omicini, A. and F. Zambonelli, *Coordination for Internet application development*, Autonomous Agents and Multi-Agent Systems **2** (1999), pp. 251–269.

[19] Petri, C. A., "Kommunikation mit Automaten," Ph.D. thesis, Institut für Instrumentelle Mathematik, University of Bonn, Bonn, Germany (1962).

[20] Plotkin, G., *A structural approach to operational semantics*, Technical Report DAIMI FN-19, Department of Computer Science, Aarhus University, Denmark (1991).

[21] Pnueli, A. and E. Shahar, *Liveness and acceleration in parameterized verification*, in: *Computer Aided Verification*, LNCS **1855**, Springer-Verlag, 2000 pp. 328–343.

[22] Ricci, A., A. Omicini and E. Denti, *Virtual enterprises and workflow management as agent coordination issues*, International Journal of Cooperative Information Systems **11** (2002), pp. 355–379.

[23] Viroli, M., *Verifying properties of coordination by well-structured transition systems*, Electronic Notes in Theoretical Computer Science **97** (2004), pp. 67–96.

[24] Viroli, M. and A. Omicini, *Coordination as a service: Ontological and formal foundation*, Electronic Notes in Theoretical Computer Science **68** (2003), pp. 457–482.

[25] Wegner, P., *Why interaction is more powerful than computing*, Communications of the ACM **40** (1997), pp. 80–91.